Free Transactions with Rio Vista

David E. Lowell and Peter M. Chen

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
{dlowell,pmchen}@eecs.umich.edu
http://www.eecs.umich.edu/Rio

Abstract: Transactions and recoverable memories are powerful mechanisms for handling failures and manipulating persistent data. Unfortunately, standard recoverable memories incur an overhead of several milliseconds per transaction. This paper presents a system that improves transaction overhead by a factor of 2000 for working sets that fit in main memory. Of this factor of 2000, a factor of 20 is due to the Rio file cache, which absorbs synchronous writes to disk without losing data during system crashes. The remaining factor of 100 is due to Vista, a 720-line, recoverable-memory library tailored for Rio. Vista lowers transaction overhead to 5 µsec by using no redo log, no system calls, and only one memory-to-memory copy. This drastic reduction in overhead leads to a overall speedup of 150-556x for benchmarks based on TPC-B and TPC-C.

1. Introduction

Any application that modifies a file takes a risk—a crash during a series of updates can irreparably damage permanent data. Grouping a series of updates into an *atomic transaction* addresses this problem by ensuring that either all the updates are applied, or none are [Gray93].

Transactions are acclaimed widely as a powerful mechanism for handling failures. Transactions simplify programming by restricting the variety of states in which a crash can leave a system. Recoverable memory provides atomic updates and persistence for a region of virtual memory [Satyanarayanan93]. Recoverable memory allows programs to manipulate permanent data structures safely in their native, in-memory form—programs need not convert between persistent and non-persistent formats [Atkinson83].

Unfortunately, while transactions are useful in both kernel and application programming, their high overhead prevents them from being used ubiquitously to manipulate all persistent data. Committing a transaction has traditionally required at least one synchronous disk I/O. This several-millisecond overhead has persuaded most systems to give up the precise failure semantics of transactions in favor of faster but less-precise behavior. For example, many file systems can lose up to 30 seconds of recently committed data after a crash and depend on ad-hoc mechanisms such as ordered writes and consistency checks to maintain file system integrity.

The main contribution of this paper is to present a system that reduces transaction overhead by a factor of 2000. We believe the resulting 5 µsec overhead makes transactions cheap enough that applications and kernels can use them to manipulate all persistent data. Two components, Rio and Vista, combine to enable this remarkable speedup.

The Rio file cache is an area of main memory that survives operating system crashes [Chen96]. Combined with an uninterruptible power supply, Rio provides persistent memory to applications that can be used as a safe, in-memory buffer for file system data. Existing recoverable-memory libraries such as RVM [Satyanarayanan93] can run unchanged on Rio and gain a 20-fold increase in performance.

Vista is a user-level, recoverable-memory library tailored to run on Rio. Because Vista assumes its memory is persistent, its logging and recovery mechanisms are fast and simple. The code for Vista totals 720 lines of C, including the basic transaction routines, recovery code, and persistent heap management. Vista achieves a 100-fold speedup over existing recoverable-memory libraries, even when both run on Rio. We expect Vista performance to scale well with faster computers because its transactions use no disk I/Os, no system calls, and only one memory-to-memory copy—factors that have been identified repeatedly as bottlenecks to scaling [Ousterhout90, Anderson91, Rosenblum95].

2. Related Work

Out of the vast literature on transaction processing, Vista is related most closely to RVM and a number of persistent stores.

RVM is an important, widely referenced, user-level library that provides recoverable memory [Satyanarayanan93]. RVM provides a simple, lightweight layer that handles atomicity and persistence. To keep the library simple and lightweight, the designers of RVM did not support other transactional properties, such as serializability and nesting, arguing that these could be better provided as independent layers on top of RVM [Lampson83].

This research was supported in part by NSF grant MIP-9521386, Digital Equipment Corporation, and the University of Michigan. Peter Chen was also supported by an NSF CAREER and Research Initiation Award (MIP-9624869 and MIP-9409229).

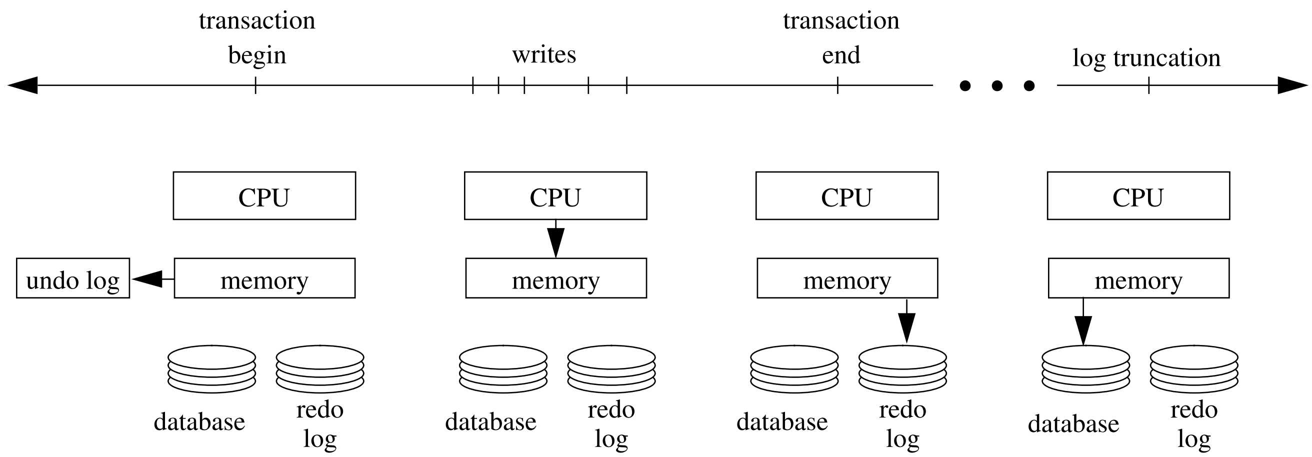


Figure 1: Operations in a Typical Recoverable Memory System. Recoverable memory systems perform up to three copies for each transaction. They first copy the before-image to an in-memory region called the undo log, which is used to support user-initiated aborts. When the transaction commits, they reclaim the space in the undo log and synchronously write the updated data to an on-disk region called the redo log. After many transactions commit and the redo log fills up, they truncate the log by propagating new data to the database on disk and reclaiming space in the redo log.

Figure 1 shows the steps involved in a simple RVM transaction. After declaring the beginning of the transaction, the program notifies RVM of the range of memory the transaction may update. RVM copies this range to an in-memory region called the undo log. The program then writes to memory using normal store instructions. If the transaction commits, RVM reclaims the space in the undo log and synchronously writes the updated data to an on-disk region called the redo log. If the user aborts the transaction, the undo log is used to quickly reverse the changes to the in-memory copy of the database. After many transactions commit and the redo log fills up, RVM truncates the log by propagating new data to the database on disk and reclaiming space in the redo log. After a process or system crash, RVM recovers by replaying committed transactions from the redo log to the database.

In addition to providing atomicity and durability, writing data to the redo log accelerates commit, because writing sequentially to the redo log is faster than writing to scattered locations in the database. Writing to the redo log in this manner is known as a *no-force* policy, because dirty database pages need not be forced synchronously to disk for every transaction [Haerder83]. This policy is used by nearly all transaction systems.

Note from Figure 1 that RVM performs up to three copy operations for each transaction. RVM performs two copies during the transaction: one to an in-memory undo log and one to an on-disk redo log. Log truncation adds at least one additional copy for each modified datum in the log.

Persistent stores such as IBM 801 Storage [Chang88] and ObjectStore [Lamb91] provide a single-level storage interface to permanent data [Bensoussan72]. Like RVM, most persistent stores write data to a redo log to avoid forcing pages to the database for each transaction.

RVM and persistent stores build and maintain an application's address space differently. RVM *copies* the database into the application's address space using `read` and maintains it using `read` and `write`. In contrast, persistent stores *map* the database into the application's address space and depend on the VM system to trigger data movement. We discuss the implications of these two choices in Section 3.

Vista differs from current recoverable memories and persistent stores in that Vista is tailored for the reliable memory provided by Rio. Tailoring Vista for Rio allows Vista to eliminate the redo log and its accompanying two copy operations per transaction. Vista also eliminates expensive system calls such as `fsync` and `msync`, which other systems must use to flush data to disk.

Having placed our work in context, we next describe the two systems that combine to reduce transaction overhead.

3. The Rio File Cache

Like most file caches, Rio buffers file data in main memory to accelerate future accesses. Rio seeks to protect this area of memory from its two common modes of failure: power loss and system crashes [Chen96]. While systems can protect against power loss in a straightforward manner (by using a $100 UPS, for example [APC96]), software errors are trickier. Rio uses virtual memory protection to prevent operating system errors such as wild stores from corrupting the file cache during a system crash. This protection does not significantly affect performance. After a crash, Rio writes the file cache data in memory to disk, a process called *warm reboot*.

To verify Rio's reliability against software crashes, Chen et al. crashed the operating system several thousand times using a wide variety of randomly chosen programming bugs. Their results showed that Rio is even more reliable than a file system that writes data immediately through to disk. To further establish Rio's ability to make memory as safe as disk, the Rio team has been storing their home direc-

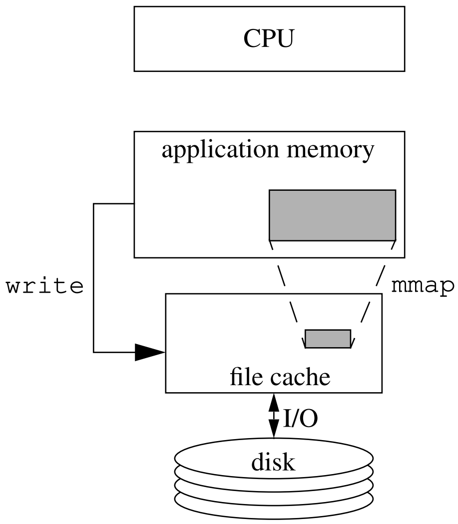


Figure 2: Using the Rio File Cache. Applications may modify persistent data in the Rio file cache in two ways: `write` and `mmap`. Most applications use `write`, which copies data from an application buffer to the file cache. `mmap` maps a portion of the file cache directly into the application's address space. Once this mapping is established, applications may use store instructions to manipulate file cache data directly; no copying is needed.

tories, source trees, and mail on Rio for a year with no loss of data.

Rio improves both performance and reliability over current systems. Under normal operation, Rio improves performance by obviating the need for synchronous I/O and by using an infinite write-back delay rather than the more common 0-30 second delay. The longer delay allows more data to be deleted or overwritten in the file cache, and hence fewer writes need to be propagated to disk. When the system crashes, Rio improves reliability by preserving data waiting to be written to disk.

There are two main ways systems can use the reliable memory Rio provides: `write` and `mmap` (Figure 2).

Applications commonly modify data in the file cache using the `write` system call. This explicit I/O operation copies data from a user buffer to the file cache. In Rio, each `write` also changes twice the protection on the modified page. To guarantee immediate permanence, applications running on non-Rio systems must use `fsync` or other synchronous I/O to force new data to disk, or they must bypass the file cache and write directly to the raw disk. On Rio, the data being written is permanent as soon as it is copied into the file cache.

In contrast to `write`, `mmap` maps a portion of the file cache directly into the application's address space. Once this mapping is established, applications can use store instructions to manipulate the file cache data directly; no copying is needed. Applications using `mmap` on non-Rio systems must use `msync` to ensure that newly written data is stored on disk. On Rio, however, data from each individual store instruction is persistent immediately; no `msync` is needed. This allows each store instruction to be its own atomic, durable transaction, a capability that is feasible only with some form of reliable memory. Vista builds on this layer to group several memory operations into a single, durable, atomic action.

Each method of modifying the file cache has advantages and disadvantages. `mmap` has significantly lower overheads, especially on Rio. Stores to a mapped region incur no system calls, while writes require one system call. In addition, `write` requires an extra memory-to-memory copy, which can lead to keeping each datum in memory twice.

The main advantage of `write` is that it may isolate the file cache from application errors. The protection mechanism in Rio focuses only on kernel errors. Users are still responsible for their own errors and can corrupt their file data through erroneous `write` calls or store instructions. Because store instructions are much more common than calls to `write`, applications may damage the file cache more easily using `mmap` [GM92]. While other systems, such as Mach's Unix server [Golub90], have mapped files into the application address space, we are aware of only two papers that quantify the increased risk [Chen96, Ng97]. Both papers indicate that (1) the risk of corruption increases only marginally when mapping persistent data into the address space and (2) memory protection can be used to further reduce this risk to below that of using a `write` interface. The first study was done on the Rio file cache and is discussed earlier in this section. The second study compares the reliability of `mmap` and `write` in the Postgres database and finds little difference. Users can enhance reliability by applying virtual memory techniques to their address space [Sullivan91, Chen96]. They can also use transactions to reduce the risk of corrupting permanent data structures in the event of a crash.

4. The Vista Recoverable Memory

The Rio file cache automatically intercepts writes that would otherwise need to be propagated synchronously to disk. For example, when RVM runs on Rio, writes to the redo log and database are not forced further than the file cache. Short-circuiting synchronous writes in this manner accelerates RVM by a factor of 20. However, it is possible to improve performance by another factor of 100 by tailoring a recoverable-memory library for Rio.

Vista is just such a recoverable-memory library. While Rio accelerates existing recoverable memories by speeding up disk operations, Vista can eliminate some operations entirely. In particular, Vista eliminates the redo log, two of the three copies in Figure 1, and all system calls. As a result, Vista runs 100 times as fast as existing recoverable memories, even when both run on Rio. Because Vista is lightweight (5 µsec overhead for small transactions) and simple (720 lines of code), it is an ideal building block for higher layers. Like RVM, Vista provides only the basic transaction features of atomicity and persistence; features such as concurrency control, nested transactions, and distribution can be built above Vista. Also like RVM, Vista is targeted for applications whose working set fits in main memory.

Figure 3 shows the basic operation of Vista. After calling `vista_init` and `vista_map` to map the database into the address space, applications start a transaction with

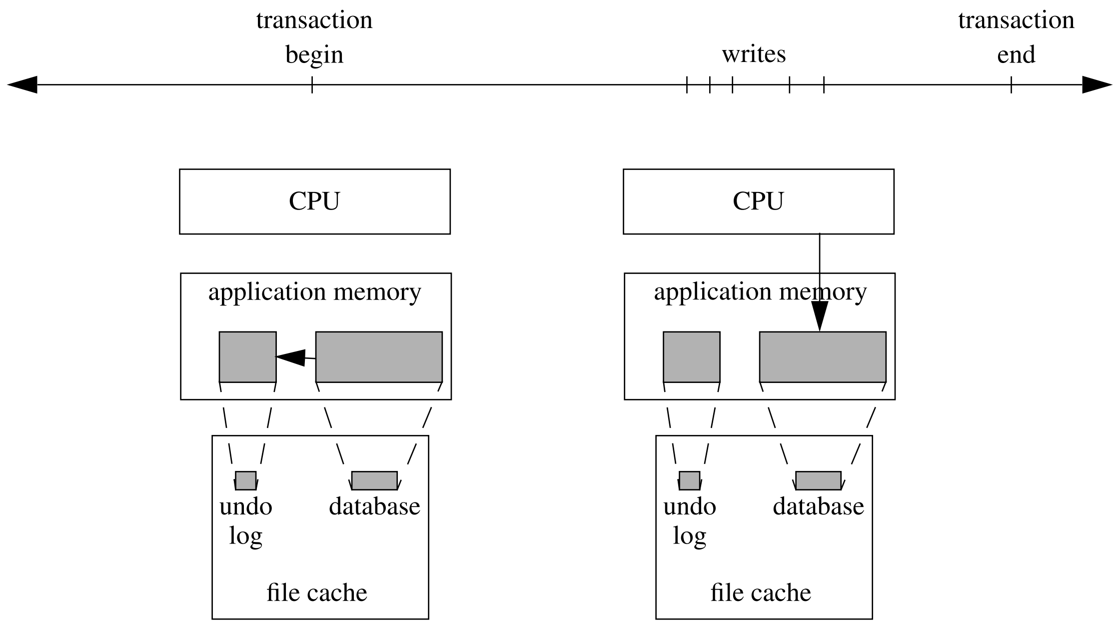


Figure 3: Operations in Vista. Vista's operation is tailored for Rio. The database is demand-paged into the file cache and mapped into the application's address space. At the start of a transaction, Vista logs the original image of the data to an undo log, which is kept in the persistent memory provided by Rio. During the transaction, the application makes changes directly to the persistent, in-memory database. At the end of the transaction, Vista simply discards the undo log.

`vista_begin_transaction`. As with RVM, applications call `vista_set_range` to declare which area of memory a transaction will modify. We insert these calls by hand, but they could also be inserted by a compiler [OToole93]. Vista also has an option to automatically generate the range of modifications on a page granularity by using the virtual memory system [Lamb91]; however this is much slower than calling `vista_set_range` explicitly. Vista saves a before-image of the data to be modified in an undo log. Like the main database, the undo log resides in mapped persistent memory provided by Rio.

After invoking `vista_set_range` one or more times, the transaction modifies the database by storing directly to its mapped image. Because of Rio, each of these stores is persistent. The transaction commits by calling `vista_end_transaction`. This function simply discards the undo log for the transaction. The transaction may abort by calling `vista_abort_transaction`. This function copies the original data from the undo log back to the mapped database.

Rio and Vista cooperate to recover data after a system crash. During reboot, Rio writes the contents of the file cache back to disk, including data from any Vista segments that were mapped at the time of the crash. Each time a Vista segment is mapped, Vista inspects the segment and determines if it contains uncommitted transactions. Vista rolls back the changes from these uncommitted transactions in the same manner as user-initiated aborts. Recovery is idempotent, so if the system fails during recovery, Vista needs only to replay the undo log again.

Designing a transaction library with Rio's persistent memory in mind yields an extremely simple system. Vista totals only 720 lines (not including comments), including the recovery code, begin/end/abort transaction, two versions of `vista_set_range` (explicit calls and VM-generated), and the persistent memory allocator described below. Several factors contribute to Vista's simplicity:

- All modifications to the mapped space provided by Rio are persistent. This enables Vista to eliminate the redo log and log truncation. Satyanarayanan, et al. report that the log truncation was the most difficult part of RVM to implement, so eliminating the redo log reduces complexity significantly [Satyanarayanan93].
- Recovery is simplified considerably without a redo log [Haerder83]. Checkpointing and recovery code are often extremely complex parts of transaction systems. In contrast, Vista does not need to checkpoint, and its recovery code is fewer than 20 lines.
- High-performance transaction-processing systems use optimizations such as group commit to amortize disk I/Os across multiple transactions. Vista issues no disk I/Os and hence does not need these optimizations and their accompanying complexity.
- Like RVM, Vista handles only the basic transaction features of atomicity and persistence. We believe features such as serializability are better handled by higher levels of software. We considered and rejected locking the entire database to serialize all transactions. While Vista's fast response times (5 µsec overhead for small transactions) makes this practical, adopting any concurrency control scheme would penalize the majority of applications, which are single-threaded and do not need locking.

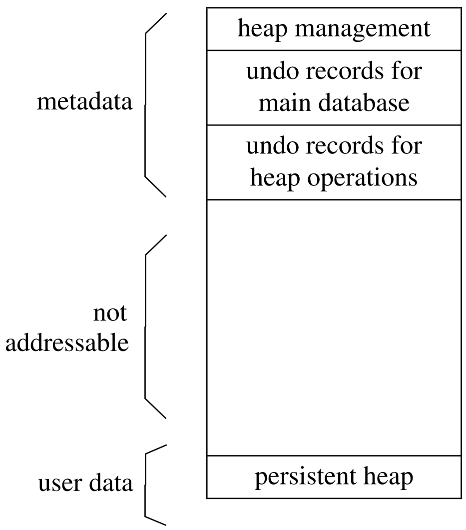


Figure 4: Memory Allocation in Vista. Metadata such as free lists and undo records are vulnerable to corruption because they are mapped into the application's address space. Vista reduces the risk of inadvertently corrupting this data by separating it from the user data.

Vista provides a general-purpose memory allocator that applications can use to dynamically allocate persistent memory. `vista_malloc` and `vista_free` can be invoked during a transaction and are automatically undone if the transaction aborts. Vista enables this feature by logically logging `vista_malloc` and `vista_free` calls. If a transaction commits, all memory that was freed in the transaction is returned to the heap. Similarly, if the transaction aborts, all memory that was allocated during the transaction is returned to the heap. As a result, aborted transactions leave the heap unchanged. Vista uses the persistent heap internally to store undo records for transactions.

Persistent heaps provide functionality similar to traditional file systems but have some unique advantages and disadvantages. Persistent heaps may be more flexible than file systems, because programs can manipulate permanent data structures in their native form—programs need not convert between persistent and non-persistent formats [Atkinson83]. For example, programs can store native memory pointers in the persistent heap, as long as the system maps the heap in a fixed location.

With the increased flexibility of heaps comes increased danger, however. Whereas metadata of a file system is inaccessible to ordinary users, the metadata describing a persistent heap is mapped into the user's address space. Vista reduces the risk of inadvertent corruption by mapping each segment's metadata into an isolated range of addresses (Figure 4). This is similar to the technique used in anonymous RPC [Yarvin93]. Other approaches to protecting the metadata could also be used, such as software fault isolation and virtual memory protection [Wahbe93].

5. Performance Evaluation

Vista's main goal is to drastically lower the overhead of atomic, durable transactions. In order to evaluate how well Vista achieves this goal, we compare the performance of three systems: Vista, RVM, and RVM-Rio (RVM running on a Rio file system).

We use RVM (version 1.3) as an example of a standard recoverable memory. We maximize the performance of RVM by storing the database and log on two raw disks. This saves the overhead of going through the file system and prevents the system from wasting file cache space on write-mostly data. The log size is fixed at 10% of the database size.

We also run RVM unmodified on a Rio file system (RVM-Rio) to show how Rio accelerates a standard recoverable memory. Storing RVM's log and data on a Rio file system accelerates RVM by allowing Rio to short-circuit RVM's synchronous writes. However, it causes the system to duplicate the database in both the file cache and application memory.

5.1. Benchmarks

We use three benchmarks to evaluate performance. We use a synthetic benchmark to quantify the overhead of transactions as a function of transaction size. We also use two benchmarks based on TPC-B and TPC-C, industry-standard benchmarks for measuring the performance of transaction-processing systems.

Each transaction in our synthetic benchmark modifies data at a random location in a 50 MB database. The size of the database was chosen to fit in main memory on all three systems. We vary the amount of data changed by each transaction from 8 bytes to 1 MB. We use the synthetic benchmark to calculate the overhead per transaction of each system. Overhead per transaction is defined as the time per transaction of a given system minus the time per transaction of a system that does not provide atomic durability. Time per transaction is the running time divided by the number of transactions.

TPC-B processes banking transactions [TPC90]. The database consists of a number of branches, tellers, and accounts. The accounts comprise over 99% of the database. Each transaction updates the balance in a random account and the balances in the corresponding branch and teller. Each transaction also appends a history record to an audit trail. Our variant of TPC-B, which we call **debit-credit**, follows TPC-B closely. We differ primarily by storing the audit trail in a 2 MB, circular buffer. We limit the size of the audit trail to 2 MB to keep it in memory and better match our target applications.

TPC-C models the activities of a wholesale supplier who receives orders, payments, and deliveries for items [TPC96]. The database consists of a number of warehouses, districts, customers, orders, and items. Our variant of the benchmark, which we call **order-entry**, uses three of the five transaction types specified in TPC-C: new-order, payment, and delivery. We do not implement the order-status and stock-level transactions, as these do not update any data and account for only 8% of the transactions. Order-entry issues transactions differently from TPC-C. In TPC-C, a number of concurrent users issue transactions at a given rate. In contrast, order-entry issues transactions serially as fast as possible. Order-entry also does no terminal I/O, as

Seagate ST31200N	
spindle speed	5411 RPM
average seek	10 ms
transfer rate	3.3-5.9 MB/s

Seagate ST15150N	
spindle speed	7200 RPM
average seek	9 ms
transfer rate	5.9-9.0 MB/s

Seagate ST12550N	
spindle speed	7200 RPM
average seek	9 ms
transfer rate	4.3-7.1 MB/s

Table 1: Disk Parameters.

we would like to isolate the performance of the underlying transaction system.

For all graphs, we run each benchmark five times, discard the best and worst runs, and present the average of the remaining three runs. The standard deviation of these runs is generally less than 1% of the mean. We run each benchmark long enough to reach steady state; hence RVM and RVM-Rio results include truncation costs.

5.2. Environment

All experiments use a 175 MHz DEC 3000/600 Alpha workstation with 256 MB of memory. We use Digital Unix V3.0 modified to include the Rio file cache.

The workstation has three data disks (Table 1). RVM and RVM-Rio use ST31200N to page out the database when it overflows memory. This disk is attached via a separate SCSI bus from ST15150N and ST12550N. RVM and RVM-Rio use ST15150N to store the redo log and ST12550N to store the database. Vista uses ST15150N to store the database.

5.3. Results

The graphs in Figure 5 compare the performance of Vista, RVM-Rio, and RVM on our three benchmarks. The synthetic benchmark measures transaction overhead as a function of transaction size, so lower values are better. Debit-credit and order-entry measure transaction throughput, so higher values are better. Note that all y-axes use log scales.

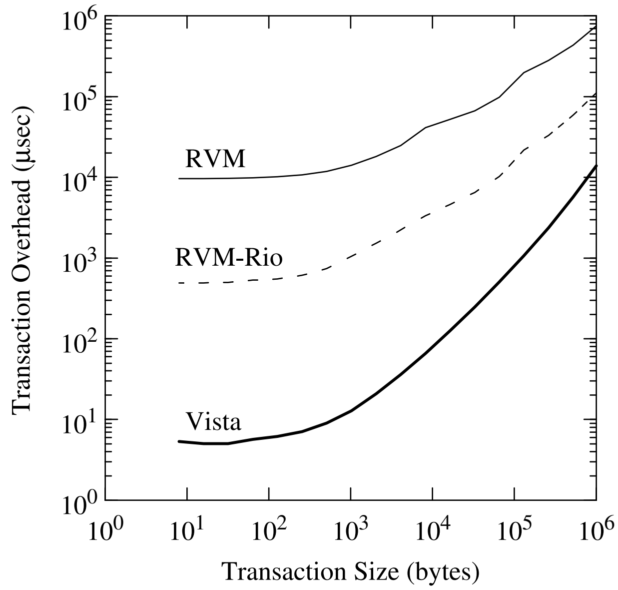
Results for the synthetic benchmark show that Vista incurs only 5 µsec of overhead for small transactions. This overhead represents the cost of beginning the transaction, saving the old data with a single `vista_set_range`, and committing the transaction. For transactions larger than 1 KB, overhead scales roughly linearly with transaction size at a rate of 17.9 µsec/KB. The increasing overhead for larger transactions is due to the need to copy more data to the undo log. Each transaction in RVM-Rio incurs a minimum overhead of 500 µsec. The higher overhead comes from the `write` and `fsync` system calls, copying to the redo log, and log truncation. Without Rio, RVM incurs one synchronous disk I/O to the redo log per transaction, as well as some portion of an I/O per transaction during log truncation. This results in an overhead of 10 ms for the smallest transaction.

Debit-credit and order-entry show results similar to the synthetic benchmarks when the database fits in memory. For debit-credit, Vista improves performance by a factor of 41 over RVM-Rio and 556 over RVM. For order-entry, Vista improves performance by a factor of 14 over RVM-Rio and 150 over RVM. These speedups are smaller than the improvement shown in the synthetic workload because the body of the debit-credit and order-entry transactions have larger fixed overheads than synthetic, and because they issue several `vista_set_range` calls per transaction.

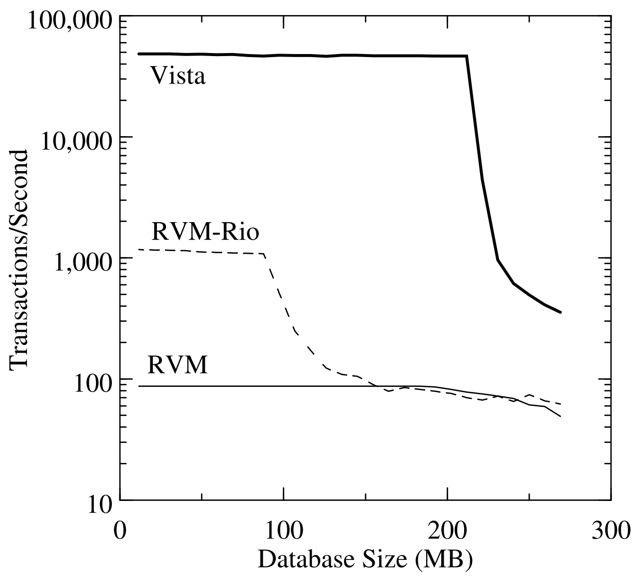
RVM and Vista both begin thrashing once the database is larger than available memory (roughly 200 MB). Note that RVM-Rio begins thrashing at half the database size of RVM and Vista due to *double buffering*. Double buffering results from frequent writes to the database file, effectively copying the database into the file cache. The result is two copies of the database: one in the process's address space and another in the Rio file cache.

Most transaction processing systems amortize the redo log I/O across multiple concurrent transactions, a technique known as *group commit* [DeWitt84]. Systems that perform this optimization wait for a number of committing transactions to accumulate, then synchronously write to disk all commit records for this group in one I/O. We implement a simple form of group commit in RVM to measure how much it improves performance. Figure 6 shows the performance of RVM when 1, 8, or 64 transactions are grouped together to share a single write to the redo log. While group commit does improve performance, it suffers from several problems. First, it does not improve response time; waiting for a group of transactions to accumulate lengthens the response time of an individual transaction. Second, throughput for large transactions can be limited by the speed of writing to the redo log. Third, group commit works only if there are many concurrent transactions. Single-threaded applications with dependent transactions cannot use group commit because earlier transactions are delayed; these applications need the fast response times provided by Vista.

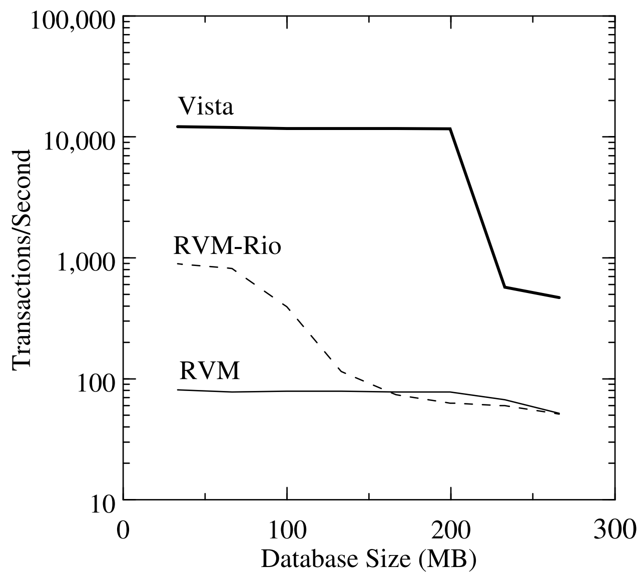
The performance improvements we have demonstrated can be broken into two components. As we expected, Rio improves performance for RVM by a factor of 11-20 by absorbing all synchronous disk writes. The biggest surprise from our measurements was how much Vista improved performance over RVM-Rio (a factor of 14-100). Once Rio removes disk writes, factors such as system calls, copies, and manipulating the redo log comprise a large fraction of the remaining overhead in a transaction. Vista does away with the redo log, all system calls, and all but one of these copies. The performance improvement resulting from the



(a) synthetic (overhead)

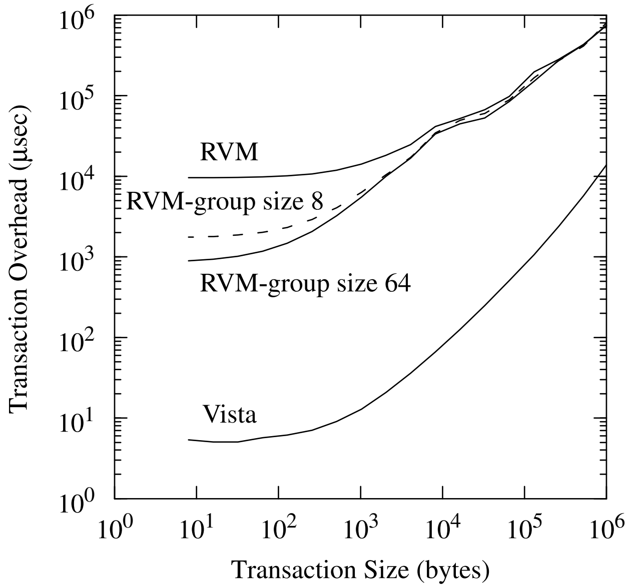


(b) debit-credit

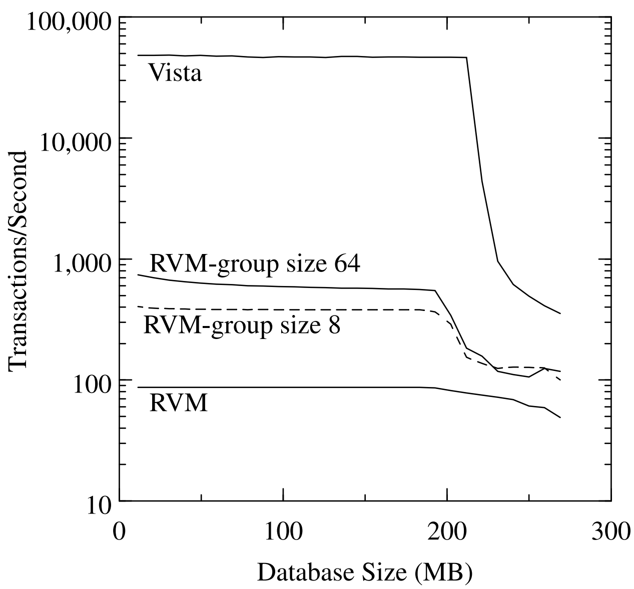


(c) order-entry

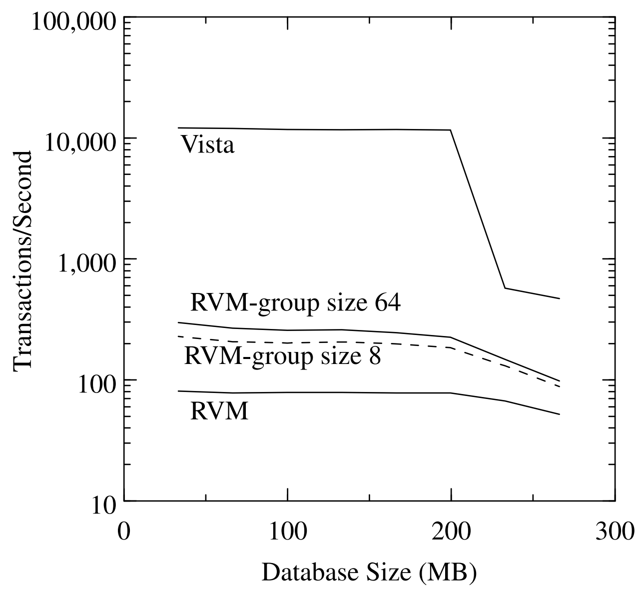
Figure 5: Performance of Vista, RVM-Rio, and RVM. This figure shows the performance of three recoverable memories on three benchmarks. Figure 5a shows the overhead per transaction for a synthetic, 50 MB database. For small transactions, Vista improves performance by a factor of 100 over RVM-Rio and a factor of 2000 over RVM. Figures 5b and 5c show the throughput for debit-credit and order-entry. Vista improves performance by a factor of 14-41 over RVM-Rio and 150-556 over RVM.



(a) synthetic (overhead)



(b) debit-credit



(c) order-entry

Figure 6: RVM Group Commit Performance. Group commit improves throughput by amortizing the synchronous write to the redo log over multiple transactions (the group commit size). However, group commit only improves throughput; response time for individual transactions actually gets slightly worse. Also, group commit works only if there are many concurrent transactions. Single-threaded applications with dependent transactions cannot use group commit because earlier transactions are delayed; these applications require fast response times. Vista's performance is shown for reference.

simplicity of Vista—Vista is less than 1/10 the size of RVM—is hard to quantify but is probably also significant.

Current architectural trends indicate that the performance advantage of Vista will continue to increase. RVM-Rio is slower than Vista because of the extra copies and system calls, while RVM is slower than Vista primarily because of synchronous disk I/Os. Many studies indicate that memory-to-memory copies, system calls, and disk I/Os will scale more slowly than clock speed [Ousterhout90, Anderson91, Rosenblum95].

6. Uses for Vista

Rio and Vista provide a very useful set of services: free persistent memory and nearly free transactions. The minimal 5 µsec overhead of a Vista transaction is small enough that it can be used even for fine-grained tasks such as swapping two pointers atomically—tasks for which disk I/Os are too expensive. We are exploring a number of ways to take advantage of Vista's fast persistence and atomicity:

- Participants in two-phase commit [Gray78] can store commit records in Vista rather than on disk to accelerate an important technique for reliable, distributed computing.
- Fault-tolerant applications can buffer network messages in Vista, then atomically send these message as part of a local transaction. Once the transaction commits, these messages will be delivered even if the system crashes.
- Software that uses a token-passing protocol can store the token in Vista and pass the token using messages within local transactions. Such a system would survive system crashes without losing the token.
- A distributed shared memory can be built on Vista to provide free persistence and transactions on the distributed memory.

7. Conclusions

The persistent memory provided by the Rio file cache is an ideal abstraction on which to build a recoverable memory. Because stores to Rio are automatically persistent, Vista can eliminate the standard redo log and its accompanying overhead, two out of the three copy operations present in standard recoverable memories, and all system calls. The resulting system achieves a performance improvement of three orders of magnitude and a reduction in code size of one order of magnitude.

Vista's minimum overhead of 5 µsec per transaction is small enough that it can be used even for fine-grained tasks such as atomically swapping two pointers. We believe Rio and Vista have brought transactions into the realm of everyday computing.[1]

1. Source code for Vista and all benchmarks used in this paper are available at http://www.eecs.umich.edu/Rio. Vista can run on non-Rio systems, however recent changes will be lost if the operating system crashes.

8. Acknowledgments

We are grateful to Jeff Chase (our shepherd), Jim Gray, Margo Seltzer, Willy Zwaenepoel, and the SOSP reviewers for providing timely insights that improved this paper significantly.

9. References

[Anderson91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the 1991 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 108–120, April 1991.

[APC96] The Power Protection Handbook. Technical report, American Power Conversion, 1996.

[Atkinson83] Malcolm Atkinson, Ken Chisholm, Paul Cockshott, and Richard Marshall. Algorithms for a Persistent Heap. *Software–Practice and Experience*, 13(3):259–271, March 1983.

[Bensoussan72] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[Chang88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[Chen96] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, October 1996.

[DeWitt84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[GM92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[Golub90] David Golub, Randall Dean, Allessandro

Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference*, 1990.

[Gray78] J. N. Gray. *Operating Systems: An Advanced Course*. Springer-Verlag, 1978. Notes on Database Operating Systems.

[Gray93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[Haerder83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[Lamb91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[Lampson83] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 1983 Symposium on Operating System Principles*, pages 33–48, 1983.

[Ng97] Wee Teck Ng and Peter M. Chen. Integrating Reliable Memory in Databases. In *Proceedings of the 1997 International Conference on Very Large Data Bases (VLDB)*, August 1997.

[OToole93] James OToole, Scott Nettles, and David Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proceedings of the 1993 Symposium on Operating Systems Principles*, pages 161–174, December 1993.

[Ousterhout90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings USENIX Summer Conference*, pages 247–256, June 1990.

[Rosenblum95] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[Satyanarayanan93] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 1993 Symposium on Operating System Principles*, pages 146–160, December 1993.

[Sullivan91] M. Sullivan and M. Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the 1991 International Conference on Very Large Data Bases (VLDB)*, pages 171–180, September 1991.

[TPC90] TPC Benchmark B Standard Specification. Technical report, Transaction Processing Performance Council, August 1990.

[TPC96] TPC Benchmark C Standard Specification, Revision 3.2. Technical report, Transaction Processing Performance Council, August 1996.

[Wahbe93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[Yarvin93] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low Latency Protection in a 64-Bit Address Space. In *Proceedings of the Summer 1993 USENIX Conference*, 1993.